Dependable Compilers: Challenges, Solutions, and Tools

He Jiang
Dalian University of Technology, China
jianghe@dlut.edu.cn
Compilers should be dependable in safety-critical areas!

According to RTCA DO-178B/C issued by the Radio Technical Commission for Aeronautics, the compiler belongs to the tools that needs to be verified.
Compiler in the aerospace should be dependable

- The compiler itself is dependable
- The compiled object is dependable
- Source code has behavioral equivalence with compiled code;
- It will not bring any security problems to the computer system during compilation.
- The executable program generated by the compiler is safe and dependable to run on the system platform.

Dependable compiler
Agenda

01 Functionality and Structure of Compiler

02 Status of Dependable Compilers

03 Strategies to Build Dependable Compilers

04 Assurance Techniques for Dependable Compilers
Functionality and Structure of Compiler

Compilers are the hub of rapid communication between human and computer!

Typical Structure of Compiler
### Functionality and Structure of Compiler

Compilers are the hub of rapid communication between human and computer!

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>C</th>
<th>C++</th>
<th>Java</th>
<th>Python</th>
<th>......</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front end</td>
<td>Lexical analysis</td>
<td>Syntax analysis</td>
<td>Semantic analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Middle end</td>
<td>Intermediate Representation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Analysis Pass</td>
<td>Transformation Pass</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Back end</td>
<td>X86 Backend</td>
<td>ARM Backend</td>
<td>......</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Targets</td>
<td>X86</td>
<td>ARM</td>
<td>MIPS</td>
<td>RISC-V</td>
<td>......</td>
</tr>
</tbody>
</table>
Modern compilers have become complex systems!

**GCC (GNU Compiler collection)**

- **Languages**: 9
- **Targets**: 63
- **Code lines**: 8.5 million+
- **Files**: 72,000+
- **Optimization**: 250+

![Structure of GCC](Uday PLDI14)
Modern compilers have become complex systems!

**LLVM** (Low Level Virtual Machine)

- **Languages**: 11
- **Targets**: 21
- **Code lines**: 6.4 million+
- **Files**: 47,000+
- **Optimizations**: 150+

Structure of LLVM
Dependable compilers for general-purpose programming languages can basically only verify a certain subset of languages, while the implementation of dependable compilers for domain-specific languages relies on the dependable compilers of general-purpose programming languages.
General-purpose Dependable Compilers

CompCert
- Side-effects out of expressions
- Type elimination
- Loop simplification
- Stack allocation
- Instruction selection
- Register allocation

Language and Architecture
- **Language**: subset of C programming language
- **Targets**: PowerPC, ARM, RISC-V and x86

Assurance Techniques
- Use **semantic preservation theory**, combined with **Coq** theorem proving tools to strictly verify the correctness of each step of compilation

CakeML
- **Language**: subset of ML programming language
- **Targets**: MIPS, ARM, RISC-V and x86

Ecosystem
- **Proof-producing synthesis**
- **Verified compiler backend**
- **Verified parsing**
- **Verified type inference**
- **Proof-producing verification-condition generation**

Use **Characteristic formula technology** for ML language to ensure the correctness of the generated code during the compilation process
**General-purpose Dependable Compilers**

**CompCert**
- **Language**: subset of C programming language
- **Targets**: PowerPC, ARM, RISC-V and x86
- **Assurance Techniques**: Use **semantic preservation theory**, combined with **Coq** theorem proving tools to strictly verify the correctness of each step of compilation.

**CakeML**
- **Language**: subset of ML programming language
- **Targets**: MIPS, ARM, RISC-V and x86
- **Use Characteristic formula technology** for ML language to ensure the correctness of the generated code during the compilation process.

---

**Dependable Compilers Ecosystem**

**Proof-producing synthesis**
- HOL functions \(\rightarrow\) CakeML AST
- CakeML AST \(\rightarrow\) machine code

**Verified compiler backend**
- Verified parsing
- Verified type inference

**Proof-producing verification-condition generation**
- CakeML AST \(\rightarrow\) Characteristic Formula
  - i.e. a 'verification condition'
### Domain-specific Dependable Compilers

<table>
<thead>
<tr>
<th>Dependable Compilers</th>
<th>Scade&amp;KCG</th>
<th>Vélus</th>
<th>L2C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Support Language and Architecture</strong></td>
<td><img src="image" alt="Scade&amp;KCG Diagram" /></td>
<td><img src="image" alt="Vélus Diagram" /></td>
<td><img src="image" alt="L2C Diagram" /></td>
</tr>
<tr>
<td><strong>Language:</strong> Lustre</td>
<td><strong>Language:</strong> Lustre</td>
<td><strong>Language:</strong> Lustre</td>
<td><strong>Language:</strong> Lustre</td>
</tr>
<tr>
<td><strong>Targets:</strong> ARM and x86</td>
<td><strong>Targets:</strong> PowerPC, ARM, RISC-V and x86</td>
<td><strong>Targets:</strong> PowerPC, ARM, RISC-V and x86</td>
<td><strong>Targets:</strong> PowerPC, ARM, RISC-V and x86</td>
</tr>
<tr>
<td><strong>Assurance Techniques</strong></td>
<td>Use <strong>strict V&amp;V process</strong> and the international standard <strong>DO-178B/C</strong> for civil aviation electronic systems to ensure the credibility of code generation</td>
<td>Translate Lustre into C language, and use <strong>Coq</strong> to strictly prove the correctness of the translation process; then employ <strong>CompCert</strong> to compile it into object code</td>
<td>Translate Lustre into C language, and use <strong>Coq</strong> to strictly prove the correctness of the translation process; then employ <strong>CompCert</strong> to compile it into object code</td>
</tr>
</tbody>
</table>
**Domain-specific Dependable Compilers**

<table>
<thead>
<tr>
<th>Dependable Compilers</th>
<th>Scade&amp;KCG</th>
<th>Vélus</th>
<th>L2C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support Language and Architecture</td>
<td><strong>Language:</strong> Lustre</td>
<td><strong>Language:</strong> Lustre</td>
<td><strong>Language:</strong> Lustre</td>
</tr>
<tr>
<td><strong>Targets:</strong> ARM and x86</td>
<td><strong>Targets:</strong> PowerPC, ARM, RISC-V and x86</td>
<td><strong>Targets:</strong> PowerPC, ARM, RISC-V and x86</td>
<td></td>
</tr>
</tbody>
</table>

**Assurance Techniques**

- Use **strict V&V process** and the international standard **DO-178B/C** for civil aviation electronic systems to ensure the credibility of code generation.

- Translate Lustre into C language, and use **Coq** to strictly prove the correctness of the translation process; then employ **CompCert** to compile it into object code.
## Domain-specific Dependable Compilers

### Dependable Compilers

<table>
<thead>
<tr>
<th>Support Language and Architecture</th>
<th>Language: Lustre</th>
<th>Targets: ARM and x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assurance Techniques</td>
<td>Use <strong>strict V&amp;V process</strong> and the international standard <strong>DO-178B/C</strong> for civil aviation electronic systems to ensure the credibility of code generation</td>
<td>Translate Lustre into C language, and use <strong>Coq</strong> to strictly prove the correctness of the translation process; then employ <strong>CompCert</strong> to compile it into object code</td>
</tr>
</tbody>
</table>

### Scade&KCG

- **Process**: Requirements Management, Design, High-Level Requirements, Low-Level Requirements, Validation, Verification

### Vélus

- **Flowchart**:
  1. Source
  2. Parsing
  3. Compressed Lustre
  4. Normalization
  5. Epigenetic Lustre
  6. Computation
  7. N-Lustre

### L2C

- **Flowchart**:
  1. Source
  2. Parsing
  3. Compressed Lustre
  4. Normalization
  5. Epigenetic Lustre
  6. Computation
  7. N-Lustre

### Language and Targets

<table>
<thead>
<tr>
<th>Domain-specific Dependable Compilers</th>
<th>Language: Lustre</th>
<th>Targets: PowerPC, ARM, RISC-V and x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scade&amp;KCG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vélus</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Agenda

01 Functionality and Structure of Compiler

02 Status of Dependable Compilers

03 Strategies to Build Dependable Compilers

04 Assurance Techniques for Dependable Compilers
Strategies to Build Dependable Compilers

Roads to build Dependable compiles

- How to define programming language?
  - User Specified
  - Existing Specification

- How to choose developing approaches?
  - Independent Development
  - Open-source Adaptation

Four possible strategies:
(1) User Specified + Independent Development
(2) User Specified + Open-source Adaptation
(3) Existing Specification + Independent Development
(4) Existing Specification + Open-source Adaptation

Specific domain-oriented language
Timeliner
LUSTRE
HAL/S
C

Front-end
Middle-end
Back-end
Formal verification

GCC
CompCert
Strategy: User Specified + Independent Development

- **Approach**
  - Front-end: design parser and IR generator
  - Middle-end: design IR optimizer
  - Backend: design backend optimizer and code generator
  - Specific domain-oriented language
  - Formal verification

- **Target**
  - Meet **Long-Term** Development Requirements

- **Pros**
  - Highly independent and trustworthy
  - Highly customizable

- **Cons**
  - Highly cost, difficult
  - Require expert knowledge
Strategy: User Specified + Open-source Adaptation

- **Approach**
  - Specific domain-oriented language
  - Formal verification
  - CompCert
  - Dependable Compilers

- **Target**
  - Meet Mid-Long-Term Development Requirements

- **Pros**
  - Ordinary independent and trusty
  - Highly customizable

- **Cons**
  - Not trusty enough
  - May lead to catastrophic consequences
Strategy: Existing Specification + Independent Development

- **Approach**
  - **Timeliner**
    - Front-end: design parser and IR generator
    - Middle-end: design IR optimizer
    - Backend: design backend optimizer and code generator
  - Formal verification
  - Dependable Compilers

- **Target**
  - Meet **Mid-Long-Term Development Requirements**

- **Pros**
  - Highly independent and trusty
  - Highly reusability

- **Cons**
  - Not customizable
  - Require expert knowledge
Strategy: Existing Specification + Open-source Adaptation

- **Approach**
  - Timeliner
  - Formal verification
  - GCC
  - CompCert
  - Dependable Compilers

- **Target**
  - Meet *Short-Term* Development Requirements

- **Pros**
  - Quick development
  - Highly reusability

- **Cons**
  - Not trusty enough
  - May lead to catastrophic consequences
Assurance Techniques for Dependable Compilers

Verification & Validation

- Using formal methods to verify the correctness of each step of a compiler

- Strong theory and completeness, but difficult for implementation and not suitable for large-scale compilers

Compiler Testing

- Using test suit or randomly generated programs to test compilers

- Easy to implement and suitable for large-scale compilers, but incomplete
Compiler Formal Verification

Formal Verification: Formally verify compiler itself

Pros:
- Completeness
- Strong theory

Cons:
- Difficult for implementation
- Hard for scalability
Compiler Translation Validation

Translation Validation: Verify semantic consistency of code

Pros:
- Easy to implement
- Scalable
- Extendable

Cons:
- Hard to define equivalence relation
- Has “false alarm”
Assurance Techniques for Dependable Compilers

### Verification & Validation

**Methods**
- Using formal methods to verify the correctness of each step of a compiler

**Keys**
- Using test suit or randomly generated programs to test compilers

**Pros & Cons**
- Strong theory and completeness, but difficult for implementation and not suitable for large-scale compilers
- Easy to implement and suitable for large-scale compilers, but incomplete
General Process of Compiler Testing

1. Test case generation
2. Test oracle
3. Test case reduction
Methods for Compiler Testing

(a) RDT

P, I

C₁, C₂, ..., Cₙ

E₁, E₂, ..., Eₙ

O₁, O₂, ..., Oₙ

If Oᵢ is different from others

Cᵢ is buggy

P: program  I: input
E: executable

(b) DOL

P, I

C-₀₁, C-₀₂, ..., C-₀ₙ

E₁, E₂, ..., Eₙ

O₁, O₂, ..., Oₙ

If Oᵢ is different from others

C-₀ᵢ is buggy

C: compiler
Oᵢ: output, 1 ≤ i ≤ n

(c) EMI

P, I

Q₁, Q₂, ..., Qₙ

C

E₀, E₁, E₂, ..., Eₙ

O₀, O₁, O₂, ..., Oₙ

If Oᵢ is different from others

C is buggy
Methods for Compiler Testing

(a) RDT

P, I

C₁ C₂ ... Cₙ

E₁ E₂ ... Eₙ

O₁ O₂ ... Oₙ

If Oᵢ is different from others
Cᵢ is buggy

P: program
I: input
E: executable

(b) DOL

P, I

C-ο₁ C-ο₂ ... C-οₙ

E₁ E₂ ... Eₙ

O₁ O₂ ... Oₙ

If Oᵢ is different from others
C-οᵢ is buggy

C: compiler
Oᵢ: output, 1 ≤ i ≤ n

(c) EMI

P, I

Q₁ Q₂ ... Qₙ

C

Eᵢ

Oₚ O₁ O₂ ... Oₙ

Compare Oₚ ≡ Oᵢ
C is buggy

If Oᵢ is different from others

C is buggy

E: executable
O: output
Methods for Compiler Testing

(a) RDT

(b) DOL

(c) EMI

P, I

C_1 C_2 ... C_n

E_1 E_2 ... E_n

O_1 O_2 ... O_n

Compare O_1, O_2, ... O_n

If O_i is different from others

C_i is buggy

P: program
I: input
E: executable

C: compiler
O_i: output, 1 ≤ i ≤ n

O_p

Q_1 Q_2 ... Q_n

Compare O_p ≡ O_i

If O_p is different from others

C is buggy

C_o1 C_o2 ... C_on

E_1 E_2 ... E_n

O_1 O_2 ... O_n

Compare O_1, O_2, ... O_n

If O_i is different from others

C_o i is buggy
Conclusion

• Dependable compilers are essential for safety-critical scenarios

• For mini compilers:
  - Compcert and CakeML are general-purpose dependable compilers
  - Scade&KCG, Vélus, and L2C are domain-specific dependable compilers

• For regular compilers:
  - Compiler testing is now the main way to achieve dependable compilers
  - 3 main stream compiler testing methods: RDT, DOL, EMI
Many thanks!

He Jiang
jianghe@dlut.edu.cn